On the Modularity of Feature Interactions

Chang Hwan Peter Kim
Department of Computer Sciences
The University of Texas at Austin
chpkim@cs.utexas.edu

Christian Kästner
School of Computer Science
University of Magdeburg, Germany
kaestner@iti.cs.uni-magdeburg.de

Don Batory
Department of Computer Sciences
The University of Texas at Austin
batory@cs.utexas.edu

Abstract

Feature modules are the building blocks of programs in *software product lines (SPLs)*. A foundational assumption of feature-based program synthesis is that features are composed in a predefined sequence called a *natural order*. Recent work on virtual separation of concerns reveals a new model of feature interactions that shows that feature modules can be quantized as compositions of smaller modules called *derivatives*. We present this model and examine some of its consequences, namely, that (1) a given program can be reconstructed by composing features in any order, and (2) the contents of a feature module (as expressed as a composition of derivatives) is determined automatically by a feature order. We show that different orders allow one to adjust the contents of a feature module to isolate and study the impact of interactions that a feature has with other features. We also show the utility of generalizing *safe composition (SC)*, a basic analysis of SPLs that verifies program type-safety, to demonstrate that every legal composition of derivatives (and thus any composition order of features) produces a type-safe program, which is a much stronger SC property.

Categories and Subject Descriptors D.2.11 Software Architectures: Languages (e.g., description, interconnection, definition)

General Terms Design

Keywords: derivatives, feature interactions, feature oriented software development, lifters, safe composition .

1. Introduction

Software product lines (SPLs) is a paradigm for the systematic and efficient creation of products. Features are increments in functionality that differentiate programs in an SPL. *Feature Oriented Software Development (FOSD)* is the study of feature modularity and the synthesis of programs in SPLs by composing feature modules [2][6][11]. A foundational assumption of FOSD, here called *natural order*, is that features are composed in a fixed and predefined sequence [5].

Natural order originates from layered designs and incremental development. Starting with a simple program, there is a natural progression in which more functionality is added by leveraging previously implemented functionality. Each increment is a feature, and a natural order is the sequence in which features are composed. Natural order permeates results in FOSD. For example, feature models can be defined as GenVoca grammars [5]: tokens are features and sentences define the products of an SPL as ordered compositions of features [7].

We and others have noticed that refactoring legacy applications into a composition of features reveals curious results [3]. Namely, when two or more people decompose the same legacy application using the same set of features, the resulting feature modules are always different. That is, the module for feature `F` in one decomposition can be quite different from the module for `F` in another decomposition. This can be explained, in part, by features having slightly different meanings, which is to be expected when features have informal definitions. But this is not the full explanation.

Looking closer, we discovered that different decompositions frequently composed features in *different* orders. That is, the same program `P` has multiple decompositions, e.g., `P=A•B•C` and `P=C'•B'•A'`, where corresponding feature modules (`A` and `A'`, `B` and `B'`, and `C` and `C'`) are different but seem "natural" as they captured similar concerns. The contents of a feature module in one decomposition are scattered and tangled in feature modules of other decompositions (e.g., the contents of `A` are scattered among `A'`, `B'`, `C'`), a phenomenon known as the "Tyranny of Dominant Decomposition" and *Multi-Dimensional Separation of Concerns (MDSoC)* [22]. This raises intriguing questions: why are there multiple feature decomposition/composition orders in a domain and not just one canonical order? And more challenging: can we automatically translate the definition of feature modules of one order into the feature modules of another?

We need to clarify an important point: it is well-known the order in which feature modules are composed matters: different programs can result and consequently, performance and behavior may change. *Our results do not change this: we are studying a different problem.* Given two (or more) decompositions of *exactly* the same program `P=A•B•C` and `P=C'•B'•A'`, where typically `A≠A'`, `B≠B'`, and `C≠C'`, this paper explains this phenomenon and shows how a set of feature modules `{A,B,C}` of one composition can be automatically translated into feature modules `{A',B',C'}` of another.

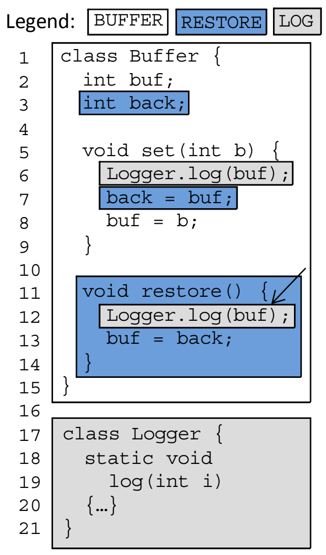
Our research requires models of feature interactions, where feature modules are quantized as compositions of smaller modules called *lifters* [21] or *derivatives* [17]. Existing models of feature interactions assume a natural order, and unfortunately cannot be used to answer our challenge questions. However, recent work on virtual separation of concerns [12] reveals a new model of feature interactions that can. In this paper, we present the *Tree Model (TM)* of feature interactions that allows us answer the questions posed above.

The Tree Model is quite different from its predecessors, and will likely impact many areas of FOSD research. One particular area is *safe composition (SC)*, a basic analysis of SPLs that proves that every legal composition of feature modules produces a type-safe program [24]. We generalize SC to show that every legal composition of derivatives also produces a type-safe program. Doing so shows that *any* composition order of features is type-safe, a much stronger SC property. We also extend SC to check whether or not a layered design is being used. We begin by describing a tool that is at the center of our work.

2. CIDE

FOSD has historically focussed on language support to define and compose feature modules [2][6][18][19]. These languages, which are centered on collaboration-based designs [23], have worked well in building feature-based SPLs from scratch. However, another common way to create an SPL is to refactor a legacy application into a composition of features modules. This process, called *Feature Oriented Refactoring (FOR)* [17], has exposed the need for significant tool support. Manually identifying the code of a feature in a legacy application and extracting the code into a module is a tedious, error-prone, and exhausting process.

Kästner, et al. have proposed *Colored Integrated Development Environments (CIDEs)* to provide this support [12]. Instead of making feature modularity an issue of programming languages, CIDE makes it a tool problem. That is, rather than extending a language with constructs to define and compose feature modules, the CIDE approach leaves languages as they are and allows programmers to paint their code. All code that is painted "blue" belongs to the `Blue` feature; all code that is painted "gray" belongs to the `Gray` feature, and so on. Gray code that appears inside blue code indicates a *structural feature interaction* — how the `Gray` feature changes the code of the `Blue` feature. Such is an example of a 1-way or 1st-order interaction. More generally, an *n*-way (or *n*th-order) interaction appears as the nesting of *n* different colors. Note our emphasis is *not* on run-time (semantic) interactions of features [8], but rather on static (syntactic) interactions that affect a feature's code.



```
class Buffer {
  int buf;
  int back;

  void set(int b) {
    Logger.log(buf);
    back = buf;
    buf = b;
  }

  void restore() {
    Logger.log(buf);
    buf = back;
  }
}

class Logger {
  static void
    log(int i)
    {…}
}
```

Figure 1. A Logged, Restorable Buffer

Consider a buffer that can be restored and logged (Figure 1). The `Buffer` class, added by the `BUFFER` feature, has clear color. The code added by the `RESTORE` feature is painted blue, and the code added by the `LOG` feature is painted gray. The interaction of the `LOG` feature with the `RESTORE` feature is indicated by nesting gray inside blue: `LOG` changes the method `restore()`, added to `BUFFER` by `RESTORE`, by logging its calls (see ↙ in Figure 1).

In CIDE, variants of a painted program can be created by eliminating features and their code. For example, a basic buffer is produced by eliminating `RESTORE` (blue) and `LOG` (gray) code. A restorable buffer is produced by eliminating the `LOG` (gray) code. And a logged buffer is produced by eliminating `RESTORE` (blue) code. Note that when the `restore()` method is eliminated, all of its code — no matter what nested colors are present — is removed.

In effect, CIDE allows programmers to separate concerns *virtually* using different colors — in its current form, CIDE has no modules (i.e. concrete representation or implementation of the concerns). But it is not hard to imagine adding editor-based views that collect all code fragments with a single color, for example, to form a virtual module. Recent work shows that language modules (such as AHEAD [6], AspectJ [15]) can be automatically projected [13]. Although projecting concrete modules is not essential for our paper, the ability to view and compose modules (concrete or virtual) is indeed useful. One possible translation of Figure 1 into AHEAD modules is shown in Figure 2. The `Super.m()` construct used in Figure 2b-c means substitute the prior definition of method `m()`; this is how method refinements are declared in AHEAD [6]. In this example, the feature modules must be composed in the order `buffer`, then `restore`, and then `log` to reproduce the buffer program of Figure 1. In this paper, we differentiate features, in `UPPERCASE`, from their modules, in `lowercase`. We explain later how to derive editor-based and language-based modules.

The effect of coloring is similar to preprocessing — if a color or feature is not selected, its code is eliminated. This orientation provides an intriguing, if not classical (*sysgen*) way to implement SPLs. Rather than composing separate feature modules, all modules are pre-composed into a single program, and particular programs of an SPL are produced by projection. Among the attractions of CIDE is that the source of a program cannot be colored arbitrarily (e.g., one cannot color ½ of an identifier or an arbitrary code fragment). Instead, only selected nodes of an *Abstract Syntax Tree (AST)* can be colored, so that removing a feature still yields an AST that satisfies the grammar of the given language [14].

The fact that CIDE represents an SPL as a single, general program, rather than as separate composable feature modules, means that the variability and commonality of the SPL must be expressible though the grammar of the base language. In CIDE's current form, an SPL must be encoded as a syntactically-valid Java program, which can rep-

```
class Buffer {
  int buf;

  void set(int b)
  { buf=b; }
}
```

(a) `buffer`

```
refines class Buffer {
  int back;

  void set(int b) {
    back=buf;
    Super.set(b);
  }

  void restore()
  { buf = back; }
}
```

(b) `restore`

```
refines class Buffer {
  void set(int b) {
    Logger.log(buf);
    Super.set(b);
  }

  void restore() {
    Logger.log(buf);
    Super.restore();
  }
}

class Logger {
  static void log(int i)
  { ... }
}
```

(c) `log`

Figure 2. A Set of AHEAD Modules for Figure 1

resent optional features easily but not necessarily alternative features (e.g. that introduce multiple variables of the same name but different types). A better understanding of how to express not only alternatives but variability in general will help improve CIDE represent SPLs, but we believe that the idea of projecting programs from a generic SPL representation is scalable.

There are various ways in which artifacts can be colored. The original version of CIDE [12] has rules for coloring that do not correspond to feature modules that can be defined in feature-based languages [2][6][18][19]. We modified CIDE to make the coloring correspond to feature-based languages. The actual differences between [12] and our version [16] in this respect are detailed in Section 5. For now, the only externally-visible property that we have changed is that every piece of code in a painted program is assigned to precisely one feature and is given that feature's color. With this clarification, we now present a general model of feature interactions that is implemented in our version, but is suitable for all versions of CIDE.

3. A Tree Model of Feature Interactions

A *structural feature interaction* represents how one feature alters the code of another feature [17][21]. We saw that CIDE indicates feature interactions by a nesting of colors. A painted program can be represented by a tree of interactions, henceforth called a *derivative tree* or simply "tree".[1] Each node of a tree represents a derivative and the tree is rooted by the empty program (denoted by `0`). If there are *k* features, there are *k* arcs leaving each node in the tree, one arc per feature. The arc for feature `F` from node `z` terminates at derivative `f\z`, which means that the derivative `f\z` encapsulates the changes that feature `F` makes to `z`, where `z` is a derivative or `0`. Note that the changes encapsulated within `f\z` may be scattered throughout a program; they need not be localized within a single method, class, or package.

In general, there is a straightforward 1-1 mapping of nested colors in a painted program to derivatives in a derivative tree. For example, the 3rd-order derivative `x\y\z` is denoted in CIDE by any code colored by feature `X` that is inside code colored by feature `Y`, which is inside code colored by feature `Z`.

The first two levels of a tree with three features (`B`, `L`, `R`) is shown in Figure 3. Note that the name of every derivative (with the exception of the root) ends in `\0`. We economize the notation by dropping "`\0`" from all derivative names henceforth in this paper.

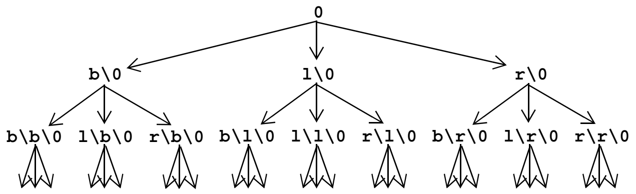


Figure 3. A Tree Model of the Buffer Program

1. This is one of several models that were developed by Kästner and Batory during Kästner's 2006-2007 visit to The University of Texas at Austin.

Each child derivative, pointed to by an arrow, changes its parent and must be composed after the parent, hence the arrow direction. Other dependencies, such as one derivative referencing a definition in another derivative, exist and can also impose ordering constraints. However, such dependencies are considered independent of a derivative tree.

By equating `B` with `BUFFER`, `L` with `LOG`, and `R` with `RESTORE`, Figure 3 is the derivative tree of our buffer program of Figure 1. In general, if there are *k* features, there are k^n nodes in a tree at level *n*. This exponential nature is a consequence of features being able to interact in *any* combination. Although there are theoretically huge numbers of derivatives, almost all are empty. There is an important special case: A feature does not interact or change itself. This means that any derivative whose name includes two or more references to the same feature (e.g., `l\l`, `l\r\l`, `r\l\l`, `l\l\l`) we equate with the identity function or null refinement or empty code (all of which are to us interchangeable). A consequence is that derivative trees with *k* features have at most *k* levels and $1+k+(k)(k-1)+(k)(k-1)(k-2)+...+k!$ derivatives.

Each derivative has a traceability link between it and AST nodes of the corresponding colored code fragments. For example, there is a mapping between Figure 3 and Figure 1 in that the derivative `b` (i.e. `b\0`) has traceability links to AST nodes colored "clear", i.e. those in lines 1, 2, 5, 8, 9, and 15 in Figure 1. And the derivative `l` to those colored gray in lines 17-21, the derivative `l\b` to those colored gray\clear in line 6, etc. Again, a *vast* number of derivatives have no associated code. The tree that is pruned of all such empty derivatives (except `0`) is shown in Figure 4, which is considerably smaller than a full tree with all derivatives.

To get a sense for actual derivative trees, we studied the *Graph Product Line (GPL)* and other SPLs used in *AHEAD Tool Suite (ATS)* [4] (Figure 5). GPL is a product-line that implements a family of graph algorithms. `jak2java`, `jampack`, `mixin`, `mmatrix`, and `unmixin` are product-lines within ATS that produce tools to manipulate Jak files (Jak is a superset of Java). `bali2jak`, `bali2javacc`, `bali2layer`, and `balicomposer` are tool product-lines that transform and compose AHEAD grammar specifications [26]. Their source code, as well as the CIDE implementation and other software tailored to this paper, can be found in [16].

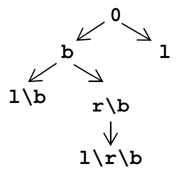


Figure 4. Pruned Derivative Tree

Instead of manually coloring these product-lines, we wrote a translator to automatically convert their composition of AHEAD feature modules into an equivalent CIDE representation.[2] For these SPLs, very few derivatives are non-empty, approximately twice the number of features. The maximum level of color nesting is at

2. The translator smashes AHEAD feature modules into feature-annotated Java files, which are then colored accordingly. The "smashing" was not easy as we had anticipated, as we had to inline chained `Super` calls (that represent refinements) to accurately nest colors. We used Eclipse JDT refactoring API in our translator to inline chained method calls. The API was not able to inline method calls whose methods exit in the middle of the method body, and we had to manually restructure these method bodies. Fortunately, these cases did not arise often.

	Lines of code	Number of Features	Number of Derivatives	Maximum Level	Average Level
GPL	1,713	17	27	3	1.67
jak2java	40,126	16	27	3	1.74
jampack	39,259	19	33	2	1.82
mixin	36,950	15	26	2	1.81
mmatrix	36,738	11	16	2	1.69
unmixin	35,817	10	14	2	1.64
bali2jak	14,780	9	15	3	1.47
bali2javacc	15,469	9	17	3	1.47
bali2layer	15,136	10	16	2	1.31
balicomposer	13,351	8	16	3	1.56

Figure 5. Size Characteristics of the SPLs Studied

most 3 and the average number of levels is about 1.5. Legacy applications that were not designed with feature modularity in mind may have many more and higher-order derivatives.

4. Mapping Trees to Programs

The code corresponding to a derivative can be represented as an editor-based module (view) or a language-based module. An editor-based view could be a window that gathers a derivative's code fragments together and that allows them to be analyzed and edited, although in our modified version of CIDE, it is simply a mechanism that provides navigation from a selected derivative to the original code fragments. A language module, such as an AHEAD module, could also express the code of a derivative. For example, for our buffer program of Figure 1, `b` (lines 1, 2, 5, 8, 9, 15) and `l` (lines 17-21) can be represented as class introductions, the "ideal" AHEAD modules for which are shown in Figure 6a-b. We explain later why we use the word "ideal"; for now it is sufficient to know that "ideal" means "simplest". Also, `l\b` (line 6), `r\b` (lines 3, 7, 11, 13 and 14), and `l\r\b` (line 12) can be represented by the "ideal" AHEAD modules in Figure 6c-e. We do not yet have a mechanism for projecting colored code fragments as AHEAD modules in our version of CIDE. Both editor-based and language-based representations should be equivalent, but as we will see later, they are not identical.

4.1 Composing Derivatives

Given that derivatives are modules (virtual or concrete), we can reconstruct a painted program incrementally by composing derivatives and observing the following rules:

- A parent derivative must always be composed before the derivative of any of its children, and
- Children of a parent can be composed in any order.

The justification of these two rules are straightforward in CIDE. The first rule reflects the nesting of colors: the outer color (i.e., the parent or base derivative) must be present before the changes of inner or nested colors (i.e., child or refinement modules) can be applied. The second rule follows from our modified CIDE coloring rule that every piece of code has exactly one color. This means that the changes made to a parent derivative by different features are

```
class Buffer {
  int buf;

  void set(int b){
    buf = b;
  }
}
```

a) `b`

```
class Logger {
  static void
    log(int i)
   {…}
}
```

b) `l`

```
refines class Buffer {
  void set(int b) {
    Logger.log(buf);
    Super(int).set(b);
  }
}
```

c) `l\b`

```
refines class Buffer {
  int back;

  void set(int b) {
    back = buf;
    Super(int).set(b);
  }

  void restore() {
    buf = back;
  }
}
```

d) `r\b`

```
refines class Buffer {
  void restore() {
    Logger.log(buf);
    Super().restore();
  }
}
```

e) `l\r\b`

Figure 6. Derivative Modules of the Buffer Program

disjoint. Stated differently, the refinements made by child derivatives of a parent derivative are commutative.

Derivatives normally reference the introductions (methods, variables) of other derivatives. As a final rule, the resulting program must be type-safe. In this section, we concentrate only on observing the first two rules. In Section 5 when we discuss safe composition, we verify the final rule.

To reproduce a program from its derivative tree, we compose derivatives of the tree obeying the above two rules. We can do this using a *reduction*, or a topological sort (parents before children), of tree nodes. Three of many possible reductions of the tree of Figure 4 are shown below as AHEAD expressions (i.e., compositions of derivatives) that synthesize the buffer program `P` of Figure 1:

$$P = l \bullet l\backslash r\backslash b \bullet r\backslash b \bullet l\backslash b \bullet b \qquad (1)$$

$$P = l\backslash r\backslash b \bullet l\backslash b \bullet l \bullet r\backslash b \bullet b \qquad (2)$$

$$P = l\backslash r\backslash b \bullet r\backslash b \bullet l\backslash b \bullet b \bullet l \qquad (3)$$

where • denotes the composition operation. `(1)` is derived from a left-to-right-depth-first reduction of the tree. `(2)` and `(3)` are derived by an algorithm we present in the next section. All of these expressions are equivalent (meaning they synthesize program `P`).

4.2 Composing Derivatives to Form Features

In practice, derivatives are too small to work with and to compose individually. We want to deal with modules that implement whole features. We know from prior work on feature interactions that feature modules are compositions of derivatives [17][21]. In this section, we show how a traversal of a derivative tree, which covers a subset of all possible reductions, yields an expression for each feature module, and that the derivatives found in a particular feature module are determined by the order in which the features are composed. We first illustrate some examples.

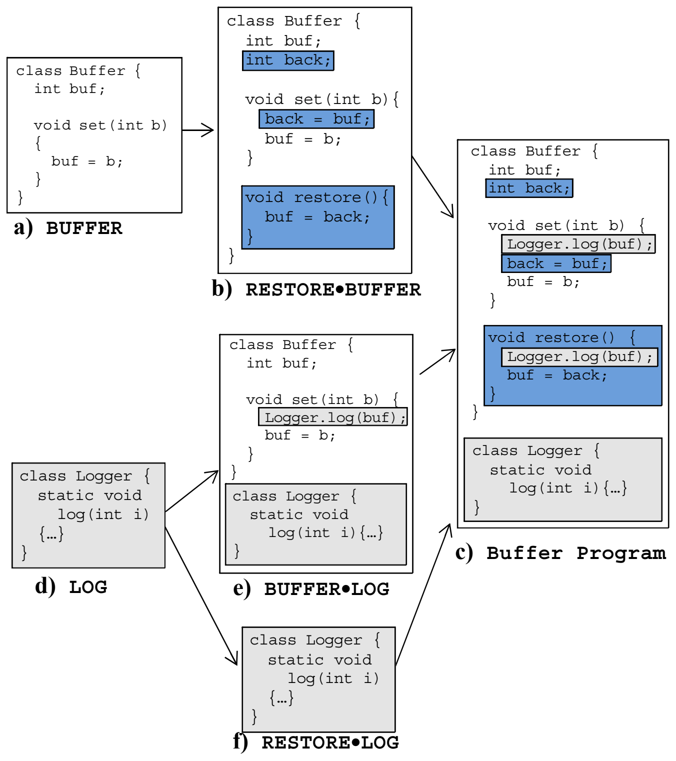
Figure 7 depicts several *stepwise development (SWD)* constructions of the buffer program. Consider the feature expression

LOG•RESTORE•BUFFER. When we add a feature, we add derivatives that are colored with that feature. Therefore, when we add **BUFFER**, only `b` derivative of Figure 6a appears, as shown in Figure 7a. Then, when we add **RESTORE**, `r\b` is added, as shown in Figure 7b. Finally, when **LOG** is added, the remaining derivatives `l`, `l\b`, and `l\r\b` are added, yielding the complete buffer program of Figure 7c. This composition of derivatives equals equation `(2)`, which was defined previously:

P = LOG • RESTORE • BUFFER
= log_2 • $restore_2$ • $buffer_2$
= (l\r\b • l\b • l) • (r\b) • (b) // See (2)

where the derivative expressions for the modules **buffer**$_2$, **restore**$_2$, and **log**$_2$ are parenthesized. We index feature modules to distinguish them from other definitions that arise later from different feature composition orders. The "ideal" AHEAD source code for these modules was presented in Figure 2 (sans subscripts). They were synthesized by evaluating the above parenthesized expressions using the derivative modules of Figure 6.

Now consider a feature expression with a different order, **RESTORE•BUFFER•LOG**. When we start with the **LOG** feature, we have a problem: we can't add all the derivatives colored **LOG** because `Buffer.set(int)` and `Buffer.restore()` have not yet been introduced. So we just add what we can, which is the `l` derivative (Figure 7d). When the **BUFFER** feature is added, we can add `l\b` and `b` (Figure 7e). Finally, when the **RESTORE** feature is added, we add `r\b` and `l\r\b`, completing the buffer program (Figure 7c). This composition of derivatives equals equation `(3)`:



a) BUFFER
```
class Buffer {
  int buf;

  void set(int b)
  {
    buf = b;
  }
}
```

b) RESTORE•BUFFER
```
class Buffer {
  int buf;
  int back;

  void set(int b){
    back = buf;
    buf = b;
  }

  void restore(){
    buf = back;
  }
}
```

c) Buffer Program
```
class Buffer {
  int buf;
  int back;

  void set(int b) {
    Logger.log(buf);
    back = buf;
    buf = b;
  }

  void restore() {
    Logger.log(buf);
    buf = back;
  }
}

class Logger {
  static void
    log(int i){…}
}
```

d) LOG
```
class Logger {
  static void
    log(int i)
  {…}
}
```

e) BUFFER•LOG
```
class Buffer {
  int buf;

  void set(int b) {
    Logger.log(buf);
    buf = b;
  }
}
class Logger {
  static void
    log(int i){…}
}
```

f) RESTORE•LOG
```
class Logger {
  static void
    log(int i)
  {…}
}
```

Figure 7. Different Stepwise Developments of the Buffer Program in CIDE

P = RESTORE • BUFFER • LOG
= $restore_3$ • $buffer_3$ • log_3
= (l\r\b • r\b) • (l\b • b) • (l) // See (3)

where the derivative expressions for modules **buffer**$_3$, **restore**$_3$, and **log**$_3$ are shown in parenthesis. The "ideal" AHEAD source code of these modules are given in Figure 8a-c. They were derived by evaluating the above expressions using the derivative modules of Figure 6.

Note: Here is the issue of "ideal" modules: AHEAD, like other collaboration-based languages, uses wrappers to extend methods. Wrappers, like advice in AOP, identify the target of a transformation in relative terms, i.e. "around" a method *as it appears at the time the transformation is applied*. So when two wrappers extend the same method, just as two pieces of advice affect the same join point, the order of application may change the outcome. For example, `l\b` (Figure 6c) and `r\b` (Figure 6d) both wrap `Buffer.set(int)`. If `r\b` is applied before `l\b`, logging occurs before restoration, which is what we want. However, if `l\b` is applied first as in `(3)`, logging occurs after restoration, which is *not* the output of CIDE.

For the buffer example, this permutation of statements is insignificant, but the issue is important in general as CIDE has no concept of wrappers; it is preprocessor. A nested colored code fragment is analogous to method inlining, which are transformations that inline calls to hook methods. Consequently, to equate AHEAD wrapper and CIDE preprocessor effects, derivatives must be made commutative by having each wrapper target a unique hook method. The `Buffer.set()` method, for example, would need an artificial hook method for logging, followed by another for restoration. These hooks will then be wrapped by **buffer**$_3$ and **restore**$_3$, respectively. Although the requirement of hook methods is not a big issue in this small example, it may

```
class Logger {
  static void log(int i)
  {...}
}
```
(a) log$_3$ **and** log$_4$

```
// empty
```
(d) restore$_4$

```
class Buffer {
  int buf;

  void set(int b) {
    Logger.log(buf);
    buf=b;
  }
}
```
(b) buffer$_3$

```
class Buffer {
  int buf;
  int back;

  void set(int b) {
    Logger.log(buf);
    back=buf;
    buf=b;
  }

  void restore() {
    Logger.log(buf);
    buf=back;
  }
}
```
(e) buffer$_4$

```
refines class Buffer {
  int back;

  void set(int b) {
    back=buf;
    Super(int).set(b);
  }

  void restore() {
    Logger.log(buf);
    buf=back;
  }
}
```
(c) restore$_3$

Figure 8. AHEAD Modules

prevent an export of larger painted programs into understandable modules. Thus, when we say "ideal" modules, we mean modules without artificial hook methods. Addressing the mismatch between painting (preprocessor) and wrapper effects is a subject of future work. This disparity does not affect the technical results in this paper, but it may make generated AHEAD or AspectJ modules inscrutable. Ultimately we think there should not be a difference between CIDE "modules" and language-based modules. Showing how this could be done is another subject for future work.

In the feature expression **RESTORE•BUFFER•LOG** above, the fact that **BUFFER** feature is able to appear *after* **LOG** feature, even though **LOG** changes **BUFFER** (**l\b**), is important because in previous models of FOSD, a feature could only change features defined previously. In the new model of FOSD based on tree of derivatives, a feature can change another feature regardless of the position, but the effect of the change (e.g. **l\b**) is only apparent when the last feature (**BUFFER**) of all the features involved in the derivative (**LOG** and **BUFFER**) appears, hence the reason **l\b** is packaged in **buffer**$_3$. Although the utility of introducing **LOG** before **BUFFER** may not be apparent at this point, it will become clear in section 4.3.2.

Finally, consider a third feature expression, **BUFFER•RESTORE•LOG**. The development of the buffer program begins with the **LOG** feature (Figure 7d). Then the **RESTORE** feature is added (Figure 7f), but notice that *nothing changes*. The reason, as in the previous example, is that **r\b** can't be added because the **Buffer** class (which is added by **BUFFER**) does not exist yet. Later, in Figure 7c, when **BUFFER** is added, the buffer class appears with all of its changes. The composition of derivatives is:

$$\begin{aligned} \mathtt{P} &= \mathtt{BUFFER} \bullet \mathtt{RESTORE} \bullet \mathtt{LOG} \\ &= \mathtt{buffer}_4 \bullet \mathtt{restore}_4 \bullet \mathtt{log}_4 \\ &= (\mathtt{l\backslash r\backslash b} \bullet \mathtt{r\backslash b} \bullet \mathtt{l\backslash b} \bullet \mathtt{b}) \bullet (\mathtt{id}) \bullet (\mathtt{l}) \end{aligned}$$

where **id** is the identity function (null refinement or empty module) and the derivative expression for the modules **buffer**$_4$, **restore**$_4$, and **log**$_4$ are in parenthesis. The "ideal" AHEAD source code for these modules is presented in Figure 8a,d,e. They were derived by evaluating the above expressions using the derivative modules of Figure 6.

All feature composition orders yield the buffer program in Figure 1, but each order induces a different expression for each feature module (e.g. **buffer**$_2$ ≠ **buffer**$_3$ ≠ **buffer**$_4$). An intuitive, general, and efficient algorithm, called the *Feature Module Composition Algorithm (FMCA)*, for computing feature modules given a feature composition order and a derivative tree, is listed in Figure 9. The algorithm traverses a derivative tree depth-first and assigns each derivative to the module of the latest feature whose symbol appears in the derivative's path. This is because we can only add a derivative (code with nested color) after all the features in its path (i.e., parent colored code) have appeared. The result of a traversal is a set of derivative expressions, one per feature, that defines how the module for that feature *for that feature order* can be synthesized.[3]

```
void computeModules(Derivative root, Expr ex)
{
    for(Derivative d:root.depthFirstTraversal())
    {
        for(Feature f: ex.lastToFirstFeatures())
        {
            if (f in d.path())
            {
                f.module().add(d);
                break;
            }
        }
    }
}
```

Figure 9. Feature Module Composition Algorithm (FMCA)

In principle, we have partial answers to the challenge problems stated in the Introduction, namely (1) For a given program, features can be composed in arbitrary orders. (2) It is possible using FMCA to automatically convert feature modules created by one composition order into feature modules of another. And (3) the fact that derivatives may appear in different feature modules for different orders explains the scattering and tangling of code noted in the Introduction.

The derivative tree is clearly the central structure behind all this. But questions remain: what is FMCA actually doing? Is there a benefit to multiple orders? What is a natural order? And how do we verify compositions of derivatives yield type-safe programs?

4.3 Perspective

4.3.1 Commuting Diagrams

Taking a step back, what we have been doing in previous sections is exploring an avenue of program synthesis that is described by a fundamental concept in mathematics called a *commuting diagram* [20]. It is a diagram of objects and morphisms (or in our case, functions) such that, when picking two objects, one can follow any path from one object to another through the diagram and obtain the same result by composition. The diagram of Figure 10a is said to commute if $g_2 \bullet f_1 = f_2 \bullet g_1$.

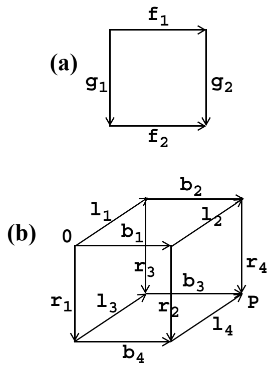


Figure 10. Commuting Diagrams

All feature composition orders that produce the buffer program (**P**) is captured by a 3-dimensional commuting diagram (Figure 10b). Each object of the diagram represents a program, and each arrow represents a feature module (a.k.a. function or transformation) that implements the **BUFFER**, **RESTORE**, or **LOG** feature. Any path from the empty program **0** to the buffer program **P** defines a unique composition order of the **BUFFER**, **RESTORE**, and **LOG** features. Different paths compose different feature modules. In general, a pro-

3. FMCA guarantees that each derivative is assigned to precisely one feature module, every parent derivative is composed before any of its child derivatives, where parent and child derivatives need not be in the same module. Child derivatives can appear in feature modules that are composed after the feature module containing the parent.

gram that is produced by composing *n* features would be represented as an *n*-dimensional commuting diagram.

FMCA is an algorithm that computes the arrows of such a diagram given a tree model of an SPL and a feature composition order. Each node in a derivative tree is a derivative (a "small arrow"); FMCA composes these small arrows to produce the composite arrows (feature modules) of Figure 10b.

4.3.2 On the Benefits of Multiple Orders

Of the SWD constructions of Figure 7, only **LOG•RESTORE•BUFFER** seems to be "natural" as it corresponds to a typical layered design a programmer might use for creating the buffer program. Other constructions are unorthodox, especially **BUFFER•RESTORE•LOG** becausethe $\texttt{restore}_4$ module is empty! But allowing these unorthodox orders, as well as natural orders, has benefits. Suppose we want to understand how the **BUFFER** feature interacts with other features in the buffer program. We would use a construction that places **BUFFER** as the final feature, such as the unorthodox **BUFFER•RESTORE•LOG**, as **BUFFER** collects the changes that it makes (i.e., `b`) to previously defined features *as well as* the changes made *to* **BUFFER** (i.e., `l\b`, `r\b`, and `l\r\b`) by previously defined features. Because each derivative has traceability links to its code fragments, **BUFFER** localizes the code fragments of lines 1-15 into an editor-based view or a language-based module, $\texttt{buffer}_4$ of Figure 8e (consisting of derivative modules of Figure 6a,c,d,e).

On the other hand, with the "natural" construction **LOG•RESTORE•BUFFER**, **BUFFER** only localizes changes to `0`, i.e. `b\0` (AST nodes of the basic, not full-fledged, `Buffer` class). But this composition order is beneficial to understand **LOG**'s interactions with other features, as the `log` module collects **LOG**'s changes to previous features (`l`, `l\b`, and `l\r\b`) and changes from the previous features to it (although **LOG** isn't changed by any feature in our example).

In general, if we want to understand how feature **F** interacts with features $X_n \ldots X_1$, we use the expression $F \bullet X_n \bullet \ldots \bullet X_1$ to obtain the module `f` for **F**. Module `f` consists of all the changes or refinements that **F** makes to $X_n \ldots X_1$ as well as the changes made to **F** by $X_n \ldots X_1$. We call this *localization* — the ability of a feature to localize the effects to and from previously defined concerns/features. Allowing multiple composition orders, or perspectives, is useful because it enables us, by simply changing the position of a feature in a composition order, to understand how that feature localizes its interactions with other features of interest. Localization is a manifestation of MDSoC.

Orders can be evaluated in ways other than localization, such as their naturalness. In Section 6, we return to the idea of natural orders.

5. Generalizing Safe Composition

Safe composition (SC) is a basic analysis of SPLs that proves that every legal composition of *feature modules* produces a type-safe program. In this section, we generalize SC in [24] to show that every legal composition of *derivatives* produces a type-safe program. Doing so shows that *any* composition order of features is type-safe, a much stronger SC property.

The version of CIDE that we used required the color of a method call to be the same as the color of the method [12] (that is, a feature introduces a method and all calls to that method). This offers a form of safe composition: all compositions of features will trivially yield a type-safe program. The coloring of Figure 1 is an example.

While this approach to coloring has merits, it is inconsistent with layered designs in general, and collaboration-based languages in particular, such as Scala [19], AHEAD [6], and Jx [18], where it is common for one layer (feature, collaboration) to introduce a method and subsequent layers to call that method. That is, it is common for method calls and method definitions (as well as references and definitions in general) to have different colors. We modified CIDE to allow common layer definitions and thus the results that we report in this paper are consistent with prior work in FOSD.[4]

Given this modification to coloring, the need for SC becomes evident. If a feature **F** calls a method that is only defined by feature **G**, then there should be no program in the SPL that has feature **F** and not **G**. If there is such a program, the SPL fails to be type-safe as one of its programs references a non-existent member. To verify that no such program exists in an SPL, feature models are used.

A feature model **FM** defines the set of all legal combinations of features; each legal combination defines a program in an SPL. It is an and-or tree where terminals represent primitive features and non-terminals represent compound features [9]. Prior work has shown how to convert a **FM** into a propositional formula ϕFM [7] or a richer formalism like first-order logic [27]. Czarnecki et al. observed that to verify no program in an SPL has property **R** (equivalently, all programs satisfy $\neg R$), it suffices to show that the following formula is unsatisfiable [10]:

$$\phi FM \wedge R$$

Thus, to verify that there is no product with feature **F** and not **G**, we need to verify that the formula $(\phi FM \wedge (F \wedge \neg G))$ is unsatisfiable. The answer can be determined efficiently using a SAT solver [7][27]. In general, SC reveals inconsistencies between feature models and implementations, but cannot point out whether it is the feature model or the implementation (or both) that needs repair.

5.1 SC with Derivatives

Prior work on SC examined a code base of feature modules to extract a set of implementation constraints expressed as propositional formulas whose terms are feature names. Simple constraints such as $F \Rightarrow G$ that should be satisfied by all programs in an SPL arise from the following sources:

- Method refinements (feature module of **F** refines a member defined in feature module of **G**)
- Reference constraints (**F** references a member defined in **G**)

4. In effect, our modification takes a step toward coloring style sheets — coloring restrictions that would conform to particular feature modularization technologies like Scala, AHEAD, etc.

More complicated constraints also arise (see [24] for a list). For the purposes of our discussion, we focus on simple constraints like $F \Rightarrow G$ as the treatment of complex constraints is the same.

The key observation in generalizing SC is that feature modules must be replaced with derivatives. Whereas before we would produce the constraint $F \Rightarrow G$ when feature module `f` references members of feature module `g`, now, if we find that derivative `a\b\f` references members introduced by derivative `x\g`, we produce the constraint `a\b\f`$\Rightarrow$`x\g`. We need to convert derivative names into formulas that reference only feature names. Derivative `a\b\f` is present in a program if and only if features `A`, `B`, and `F` are present in a program ($A \wedge B \wedge F$). Similarly, `x\g` is present if and only if both `X` and `G` are present ($X \wedge G$). Therefore, the property that all programs in an SPL must satisfy for `a\b\f`$\Rightarrow$`x\g` is:

$$A \wedge B \wedge F \Rightarrow X \wedge G$$

That is, if the features `A`, `B`, and `F` are in a program, so too must the features `X` and `G` Thus whenever we generate a constraint for derivative `d` whose name is $c_1 \backslash c_2 \backslash \ldots \backslash c_n$, we replace `d` by the formula ϕd, which is $C_1 \wedge C_2 \wedge \ldots \wedge C_n$.

Given this straightforward translation, we observed the following:

- CIDE effectively eliminates refinement constraints. Derivative `x\d`, which refines derivative `d`, maps directly to the constraint $(\phi x \wedge \phi d) \Rightarrow \phi d$ which is vacuously true. This is because a derivative's existence is associated with not just its color but all of its ancestors' colors as well. There is no need to verify such conditions.
- Derivatives with different path names, but with the same set of colors, such as `l\r\b` and `r\l\b`, have the same presence condition ($L \wedge R \wedge B$), so many redundant constraints that are generated can be eliminated.
- CIDE eliminates multiple introduction constraints. Only one definition of a member is permitted in CIDE, as is in Java; AHEAD allows multiple definitions in alternative features. CIDE's restriction means that (mutually exclusive) alternatives are difficult to express in its current form. This is yet another example where CIDE's virtual modules and AHEAD concrete modules are not quite the same.

We also realized that two new properties had to be verified. First, recall from Section 4.2 that the feature `RESTORE` could have an empty module. We do not want a composition of modules resulting in an empty program. Thus, we require that for every program in an SPL, at least one non-empty derivative be used. If $\{d_1 \ldots d_n\}$ is the set of non-empty derivatives, the property that an SPL must satisfy is:

$$\phi d_1 \vee \phi d_2 \vee \ldots \vee \phi d_n \qquad (4)$$

Second, we want to ensure that every derivative of a painted program is used in some program of an SPL. If there is a derivative `d` that appears in no product, `d` is effectively dead-code, and designers should be alerted. Again, let $\{d_1 \ldots d_n\}$ be the set of non-empty derivatives. The properties that an SPL must satisfy are:

```
¬φd1          // constraint 1
...
¬φdn          // constraint n            (5)
```

5.2 Experimental Results

We compared AHEAD SC against CIDE SC on the same set of product-lines, those listed in Figure 5 (first explained in Section 3). Due to the similarity of results, we only show the calculations for `GPL`, `jak2java`, and `bali2jak` SPLs in Figure 11a-c. Note that the new constraint types **NoEmptyProgram** and **NoUnusedDerivative** are available only in CIDE SC. Also, **Introduction** constraints, for checking overwrites of introductions, are only available in AHEAD SC.

5.2.1 Number of Constraints

Overall, the total number of unique constraints to verify is similar between both versions for all SPLs, which is expected because 1) the lack of refinement constraints in CIDE is compensated by the new constraints in CIDE and 2) SC generalization basically replaced features with derivatives, but did not change the number of constraints. There are a few interesting differences though. The number of non-unique referential constraints is considerably lower in the CIDE version. The reason is that a derivative, such as logging, typically references the parent code that it injects into, such as the data to log. Because a derivative requires its parent in CIDE by definition, these referential properties are trivially true and need not be checked. The numbers of interface constraints produced for `jak2java` are slightly different because AHEAD SC does not count refinements that add interfaces, which it should.

5.2.2 Number of Failures

CIDE SC produces constraints that are different than those produced by AHEAD SC because constraints on a single feature are now spread out across multiple derivatives. For example, in Figure 2, `LOG` requires both `RESTORE` and `BUFFER`, while it requires neither in its CIDE version, Figure 1, because there are no refinement constraints in CIDE. So the fact that the results show more referential failures in the CIDE version requires an explanation. In GPL, `PROG` $\Rightarrow$ `(WEIGHTED`$\wedge$`BASE)` because `prog` $\Rightarrow$ `(weighted\base)`. `prog` calls method `addAnEdge()`, which is defined in `weighted\base`. Upon closer inspection, it turns out that an *identical version* of `addAnEdge()` was also defined in `base`, but was overwritten by `weighted\base` when all the features in the AHEAD version were composed to produce the CIDE version. Without the multiple introductions, the referential constraint would be `PROG`$\Rightarrow$`BASE`, which the feature model satisfies. Further investigation confirmed that all the errors that were new in the CIDE version were the result of multiple introductions. Errors present in AHEAD were also present in CIDE.

While we knew there were multiple introduction *warnings* in AHEAD, which are shown in Figure 11, we did not anticipate them causing type safety *errors* in the CIDE version. In hindsight, automated translation from AHEAD version to CIDE version should have been preceded by a careful analysis of all the multiple introductions present. But this would have been difficult as any set of *intended* multiple introductions would need to be manually converted into static alternatives expressible in the Java language, as noted in the previous section. Even a seemingly simple case of rep-

	AHEAD Safe Composition		CIDE Safe Composition	
Constraint Type	No. of Constraints	No. of Failures	No. of Constraints	No. of Failures
Refinement	49	0	0	0
Referential	448	0	131	1
Introduction	2	2	unavailable	unavailable
Abstract class	0	0	0	0
Interface	0	0	0	0
NoEmptyProgram	unavailable	unavailable	1	0
NoUnusedDerivative	unavailable	unavailable	26	0
Total	499 (49 unique)	2	158 (63 unique)	1

(a) GPL

	AHEAD Safe Composition		CIDE Safe Composition	
Constraint Type	No. of Constraints	No. of Failures	No. of Constraints	No. of Failures
Refinement	155	0	0	0
Referential	7,000	1	5,753	5
Introduction	3	3	unavailable	unavailable
Abstract class	345	0	345	0
Interface	12	0	14	0
NoEmptyProgram	unavailable	unavailable	1	0
NoUnusedDerivative	unavailable	unavailable	26	0
Total	7,515 (147 unique)	4	6,139 (99 unique)	5

(b) jak2java

	AHEAD Safe Composition		CIDE Safe Composition	
Constraint Type	No. of Constraints	No. of Failures	No. of Constraints	No. of Failures
Refinement	22	0	0	0
Referential	2,378	0	1,349	1
Introduction	2	2	unavailable	unavailable
Abstract class	57	0	57	0
Interface	18	0	18	0
NoEmptyProgram	unavailable	unavailable	1	0
NoUnusedDerivative	unavailable	unavailable	14	0
Total	2,477 (62 unique)	2	1,439 (56 unique)	1

(c) bali2jak

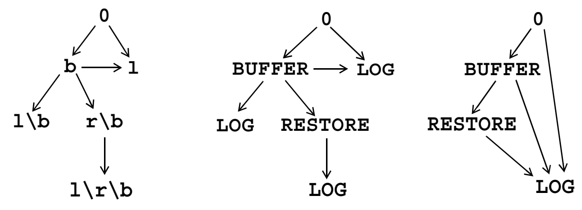
Figure 11. Safe Composition Experiment Results

resenting two methods with the same signature but different bodies as alternatives requires highly manual, if not difficult, effort. This observation illustrates once again the mismatch, or a trade-off, between language-based separation of concerns, which provides the benefits of wrapping and multiple introductions, and virtual separation of concerns, which provides an appealing environment for decomposing programs.

6. Natural Order

Although features can now be ordered in any way for localizing interactions, we can still provide a check that determines whether or not a reconstruction order of a program is as intuitive as a natural order that would have been used to construct a program from scratch. A natural order corresponds to a strict definition of layered development: a feature module can only reference and refine classes that exist (i.e., that have been defined in feature modules that have been composed previously).

A *feature dependency graph (FDG)* captures both refinement and reference relationships. To create an FDG, we start with a pruned derivative tree (Figure 4) which captures refinement relationships. We superimpose referential dependencies on the tree as shown in Figure 12a (imagine that `l` references `b`, creating `b`→`l`). We then replace each derivative with the color that introduced it, i.e., the last feature in its path, as shown in Figure 12b. Finally, we merge nodes with the same feature names and merge redundant arrows between pairs of nodes, to produce the FDG of Figure 12c. An arrow **`A`**→**`B`** in a FDG means feature **`A`** references or refines **`B`**, and a natural order corresponds to a topological sort of an FDG. A cycle, e.g. **`A`**↔**`B`**, in an FDG means that features **`A`** and **`B`** either refine or reference each other's introductions. This could be a consequence of a bad design (which could be resolved by a simple refactoring that moves members of **`A`** into **`B`**, or into an ancestor feature of both **`A`** and **`B`**, or by having **`A`** call empty stub methods which **`B`** can later refine by injecting calls to **`B`**-introduced members). A cycle **`A`**↔**`B`** could also mean features **`A`** and **`B`** are so closely coupled that they must be designed together, or that features **`A`** and **`B`** actually belong to a single feature that should never have been partitioned. All of these situations are possible, but cycles do not distinguish which of these possibilities have occurred. With cycles resolved manually, a natural order is simply a topological sort of the FDG.



a) Superimpose referential dependencies **b) Replace derivatives with colors** **c) Merge nodes and edges**

Figure 12. Feature Dependency Graph Construction

For each of the SPLs in Figure 5, we constructed an FDG, detected cycles using a standard algorithm, determined a reason for each cycle, and removed them. Then we checked whether the *nominal* order of features (i.e. the order that AHEAD used to synthesize the program) was a topological sort of its FDG. The results for five SPLs, which are representative of the rest, are shown in Figure 13.

Interestingly, we discovered a number cycles in AHEAD SPLs. One cycle, **`GSCOPE`**↔**`AST`** in **`jak2java`**, confirmed a long-held suspicion that the **`GSCOPE`** and **`AST`** layers have never been fully separated. But because they always appeared together in programs, we were never quite sure. Other cycles, such as **`PREPRO-`**

	Cycles	**Is nominal order natural after cycles are resolved?**
GPL	0	No. Reason: PROG (at index 16) depends on BENCHMARK (at index 17)
jak2java	3 (GSCOPE<->AST, J2JBASE<->KERNEL, JAVA<->KERNEL)	No. Reason: J2JAST (at index 8) depends on J2JBASE (at index 10)
jampack	5 (E.g. SORTFD <-> COMPCLASS, PREPROCESS<->COMPINT, COMMONBASE<->PREPROCESS)	No. Reason: COMPCLASSAST (at index 9) depends on COMPCLASS (at index 17)
bali2jak	7 (E.g. KERNEL<->BALI, BALI2JAK<-> REQUIREBALI2JAK, BALI<->SYNTAX)	Yes.
balicomposer	7 (E.g. REQUIRE <-> REQUIRECOMPOSER, BALI<-> COMPOSER, KERNEL<->COMPOSER)	Yes.

Figure 13. Natural Order Experiment Results

`CESS`↔`COMPINT` in `jampack`, exposed design errors. `PREPROCESS` unnecessarily references a member that was introduced by `COMPINT`, a layer that is composed after `PREPROCESS`. `COMPINT` should have refined `PREPROCESS` to add this reference. Most cycles, however, were due to a layer introducing a method, and a subsequent layer overriding that method with exactly the same definition. This is an artifact of copy-and-paste programming which was not a design error until the new SC analysis was performed.

In general, SC using derivatives exposes more errors than the original version of SC found. This is a consequence of exposing constraints among finer-grained modules (derivatives) and their relationships.

We were also surprised to discover that composition orders used in AHEAD for some time are not natural orders. For example, the `PROG` and `BENCHMARK` layers in `GPL` were permuted some time ago (`PROG` references introductions of `BENCHMARK`, and thus should be composed last). As the resulting programs didn't have compilation errors, we were unaware of the permutation.

7. Related Work

Our paper was inspired by CIDE [12] and recent work on treating aspects as program transformations and the idea of *pseudo-commutativity (PC)* [1][3]. PC was motivated by a long-held belief in the AOP community that the order in which aspects are extracted/refactored from programs doesn't matter, but the implications of ordering were not well-known. [3] showed that the order in which aspects are refactored makes a difference in their implementation, where some implementations are easy to understand and other implementations are abstruse. See [3] for a thorough list of related work in the AOP literature.

Our paper addresses the same problem for features but with important differences. First, the mechanisms for reversing the order of features in Section 4.2, FMCA, is a general and simple algorithm to implement, whereas [3] may require sophisticated transformations to map one aspect to another. Second, our work is based not on aspects but rather on feature interactions. Whether derivative trees can be used to express compositions of functional aspects (i.e., a functional aspect is an aspect that can be considered a transformation) is an open problem. However, both [3] and our work shed light on a fundamental commutativity property of composed transformations in AOP and FOSD.

Our paper was also inspired by prior work on structural feature interactions. Prehofer first proposed lifters as a way of expressing 2-way feature interactions [21]. Liu et al. generalized Prehofer's approach to allow for certain kinds of n-way interactions [17]. In particular, a feature could only modify the contents (modules) of previously composed features (called *past interactions*), and could not modify the contents of features that are subsequently composed, called *future interactions*. On the other hand, the Tree Model of Section 3 uniformly and simply expresses both past and future interactions, although having both future and past interactions lead to cycles as discussed in Section 6. Still, the Tree Model is similar to the previous models in that it can be used to study the interactions of large-scale features as the size of features/derivatives is irrelevant to the Tree Model. Although the research of this paper was limited to illustrating the interaction of small features, the notion of deriviatives need not be limited to small code fragments. Features can be of arbitrary size and feature interactions can always arise.

The Tree Model of feature interactions affects many research areas, including safe composition, of FOSD. This paper generalized the safe composition presented in [24], which is based on the notion of checking well-formedness of a feature-based model template [10]. Safe composition of SPLs is an open and active research area. For example, a formal approach to type-checking CIDE product-lines based on extended Featherweight Java was introduced recently [28]. The problem of representing alternatives is also briefly discussed there, but is largely left open as is here.

Modularizing feature interactions based on virtual separation of concerns is related to *Fluid AOP* [25]. Fluid AOP provides join point models where aspects define editor-based views of a program, through which the program can be analyzed and edited. For example, in "Gather Join Point Model", aspects localize a set of join points in a view and allow them to analyzed and edited. Features in the paper are similar to Fluid AOP aspects in that feature modules as defined by an ordering are also editor-based views that have traceability links to AST nodes in the original program. But an important difference between these two concepts is that while fluid AOP aspects are defined in terms of the static structure of a program afforded by its language, features are defined in terms of a more fundamental structure, namely, that of feature interactions.

8. Conclusions

FOSD — the study of feature modularity and feature-based program synthesis in SPLs — rests on a small number of simple ideas. Even so, the relationship between some ideas are not yet fully-understood. In this paper, we explored the relationship between feature modularity, feature interactions, and natural orders. We showed how a recent advance in SPL tools, namely CIDE, is based on the Tree Model, a new and general model of feature interactions. We explored some of its consequences. Namely that for any program, features can be composed in any order, and the reconstruction order dictates the contents of feature modules.

We explained how the contents of a program can be painted in CIDE and how different nestings of colors map to a derivative tree, an instance of the Tree Model. Each derivative encapsulates interactions among particular compositions of features. We presented FMCA, an algorithm that traverses a derivative tree, reassembles a program given a particular feature order, and also produces contents of the feature modules for that order. We observed that no single order was the best for all needs. In fact, we explained how each order achieved localization — the ability to modularize a feature's interactions with other features. Localization is a particular concrete manifestation of MDSoC.

With the Tree Model, we generalized safe composition, a basic analysis of SPLs that verifies program type-safety, to prove that every legal composition of derivatives produces a type-safe program. We evaluated our ideas by converting feature-composed programs in AHEAD into a CIDE-form to analyze derivative trees, their safe composition, and natural orders, noting where CIDE improves existing results, as well as exposing areas in which CIDE can be improved.

Among the key topics for further work is addressing the misalignment of CIDE's preprocessor effects and wrapper effects of AHEAD and other feature- and aspect-based languages, as well as the need for CIDE's improved support for alternatives (method/field overrides and their subsequent refinements). These issues lie at the intersection of tool-oriented and language-oriented solutions for FOSD. Another important topic is semantic feature interactions since features ultimately change program behaviour and are bound to interact at runtime. Lastly, a formalization of our models may reveal implicit assumptions on which our work is based. All of these topics will be a source of interesting research in the future.

Acknowledgments. We appreciate the helpful and timely comments of the referees. This work was supported by NSF's Science of Design Projects #CCF-0438786 and #CCF-0724979.

9. References

[1] S. Apel and J. Liu. "On the Notion of Functional Aspects in Aspect-Oriented Refactoring", *ADI Workshop 2006*.

[2] S. Apel, T. Leich, and G. Saake. "Aspectual Feature Modules". *IEEE TSE*, April 2008.

[3] S. Apel, C. Kästner, and D. Batory, "Program Refactoring using Functional Aspects". *GPCE 2008*.

[4] AHEAD Tool Suite, `www.cs.utexas.edu/users/schwartz/index.html`

[5] D. Batory and S. O'Malley. "The Design and Implementation of Hierarchical Software Systems with Reusable Components". *ACM TOSEM*, October 1992.

[6] D. Batory, J.N. Sarvela, and A. Rauschmayer. "Scaling Step-Wise Refinement", *IEEE TSE*, June 2004.

[7] D. Batory. "Feature Models, Grammars, and Propositional Formulas", *SPLC* 2005.

[8] M. Calder, M. Kolberg, E.H. Magill, and S. Reiff-Marganiec. "Feature Interaction: A critical Review and Considered Forecast". *Computer Networks*, January 2003.

[9] K. Czarnecki and U. Eisenecker. *Generative Programming Methods, Tools, and Applications*. Addison-Wesley, Boston, MA, 2000.

[10] K. Czarnecki and K. Pietroszek. "Verification of Feature-Based Model Templates Against Well-Formedness OCL Constraints". *GPCE 2006*.

[11] S. Trujillo, M. Azanza, and O. Diaz. "Generative Metaprogramming", *GPCE* 2007.

[12] C. Kästner, S. Apel, and M. Kuhlemann. "Granularity in Software Product Lines". *ICSE* 2008.

[13] C. Kästner, M. Kuhlemann, and D. Batory. "Automating Feature-Oriented Refactoring of Legacy Applications", *ECOOP 2006 Poster Paper.*

[14] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. "Language-Independent Safe Decomposition of Legacy Applications into Features". TR #2, School of Computer Science, University of Magdeburg, Germany, March 2008.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kirsten, J. Palm, W.G. Griswold. "An Overview of AspectJ", *ECOOP* 2001.

[16] C. H. P. Kim. Implementation accompanying GPCE 2008 submission available at `www.cs.utexas.edu/~chpkim/gpce08`

[17] J. Liu, D. Batory, and C. Lengauer. "Feature Oriented Refactoring of Legacy Applications", *ICSE* 2006.

[18] N. Nystrom, S. Chong, A.C. Myers. "Scalable Extensibility via Nested Inheritance". *OOPSLA 2004*.

[19] M. Odersky, P. Altherr, V. Cremet, I. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger. "An Overview of the Scala Programming Language". September 2004, EPFL Technical Report IC/2004/64.

[20] B. Pierce. *Basic Category Theory for Computer Scientists*, MIT Press, 1991.

[21] C. Prehofer, "Feature Oriented Programming: A Fresh Look at Objects". *ECOOP* 1997.

[22] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton. "N Degrees of Separation: Multi-Dimensional Separation of Concerns", *ICSE* 1999.

[23] Y.Smaragdakis and D. Batory. "Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs". *ACM TOSEM*, April 2002.

[24] S. Thaker, D. Batory, D. Kitchin, and W. Cook, "Safe Composition of Product Lines ", *GPCE* 2007.

[25] T. Hon and G. Kiczales. "Fluid AOP Join Point Models". *OOPSLA Companion 2006*.

[26] S. Thaker. "Design and Analysis of Multidimensional Program Structures", MA thesis, The University of Texas at Austin, 2006. Available at `ftp://ftp.cs.utexas.edu/pub/predator/SahilThesis.pdf`

[27] J. Sun, H. Zhang, and H. Wang. "Formal Semantics and Verification for Feature Modeling". *ICECCS 2005*.

[28] C. Kästner and S. Apel. "Type-checking Software Product Lines - A Formal Approach". *ASE 2008*.